A scanner or lexer transforms a string of characters into a string of tokens:
- uses a combination of deterministic finite automata (DFA);
- plus some glue code to make it work;
- can be generated by tools like flex (or lex), JFlex, ...

Tokens are defined by regular expressions:
- $\emptyset$, the empty set: a language with no strings
- $\varepsilon$, the empty string
- $a$, where $a \in \Sigma$ and $\Sigma$ is our alphabet
- $M \mid N$, alternation: either $M$ or $N$
- $M \cdot N$, concatenation: $M$ followed by $N$
- $M^*$, zero or more occurrences of $M$

where $M$ and $N$ are both regular expressions. What are $M\?\text{?}$ and $M\+	ext{?}$

We can write regular expressions for the tokens in our source language using standard POSIX notation:
- simple operators: "*", "/", "+", "-"
- parentheses: "(\text{.}, \text{\text{.}})"
- integer constants: 0|([1-9][0-9]*)
- identifiers: [a-zA-Z_][a-zA-Z0-9_]*
- white space: [\w\t\n]+
flex accepts a list of regular expressions (regex), converts each regex internally to an NFA (Thompson construction), and then converts each NFA to a DFA (see Appel, Ch. 2):

Given DFAs $D_1, \ldots, D_n$, ordered by the input rule order, the behaviour of a flex-generated scanner on an input string is:

```
while input is not empty do
    $s_i :=$ the longest prefix that $D_i$ accepts
    $l := \max\{|s_i|\}$
    if $l > 0$ then
        $j := \min\{i : |s_i| = l\}$
        remove $s_j$ from input
        perform the $j$th action
    else (error case)
        move one character from input to output
    end
end
```

In English:
- The longest initial substring match forms the next token, and it is subject to some action
- The first rule to match breaks any ties
- Non-matching characters are echoed back

Why the “longest match” principle?
Example: keywords

```
[ \t]+
    /* ignore */;
... import
    return tIMPORT;
... [a-zA-Z_][a-zA-Z0-9_]* {
    yylval.stringconst = (char *)malloc(strlen(yytext)+1);
    printf(yylval.stringconst, "%s", yytext);
    return tIDENTIFIER;
}
```

Want to match “importedFiles” as tIDENTIFIER(importedFiles) and not as tIMPORT tIDENTIFIER(edFiles).
Because we prefer longer matches, we get the right result.

Why the “first match” principle?
Again — Example: keywords

```
[ \t]+
    /* ignore */;
... continue
    return tCONTINUE;
... [a-zA-Z_][a-zA-Z0-9_]* {
    yylval.stringconst = (char *)malloc(strlen(yytext)+1);
    printf(yylval.stringconst, "%s", yytext);
    return tIDENTIFIER;
}
```

Want to match “continue foo” as tCONTINUE tIDENTIFIER(foo) and not as tIDENTIFIER(continue) tIDENTIFIER(foo).
“First match” rule gives us the right answer:
When both tCONTINUE and tIDENTIFIER match, prefer the first.
When “first longest match” (flm) is not enough, look-ahead may help.

FORTRAN allows for the following tokens:

- .EQ., 363, 363., .363

flm analysis of 363.EQ.363 gives us:

`tFLOAT(363) E Q tFLOAT(0.363)`

What we actually want is:

`tINTEGER(363) tEQ tINTEGER(363)`

`flex` allows us to use look-ahead, using `/`

`363/ .EQ. return tINTEGER;`

Another example taken from FORTRAN:

Fortran ignores whitespace

1. D05I = 1.25 ~ D05I=1.25
   in C: do5i = 1.25;

2. D0 5 I = 1,25 ~ D05I=1,25
   in C: for(i=1;i<25;++i){...}

   (5 is interpreted as a line number here)

Case 1: flm analysis correct:

`tID(D05I) tEQ tREAL(1.25)`

Case 2: want:

`tDO tINT(5) tID(I) tEQ tINT(1) tCOMMA tINT(25)`

Cannot make decision on tDO until we see the comma!

Look-ahead comes to the rescue:

`DO/({letter}|{digit})*=({letter}|{digit})*,`

Using `flex` to create a scanner is really simple:

```
$ emacs print_tokens.l
$ flex print_tokens.l
$ gcc -o print_tokens lex.yy.c -lfl
```

When input `a*(b-17) + 5/c`:

```
$ echo "a*(b-17) + 5/c" | ./print_tokens
```

our `print_tokens` scanner outputs:

```
identifier: a
times
left parenthesis
integer constant: 17
right parenthesis
white space, length 1
plus
white space, length 1
integer constant: 5
div
identifier: c
white space, length 1
```

You should confirm this for yourself!
**Count lines and characters:**

```c
#include <stdlib.h> /* for atoi */
#include <stdio.h> /* for printf */

#include "stdlib.h" /* for atoi */
#include "stdio.h" /* for printf */

int lines = 0, chars = 0;

lines++; chars++;

main () {
    yylex ();
    printf (#lines = %i, #chars = %i\n", lines, chars);
}
```

**Remove vowels and increment integers:**

```c
#include <stdlib.h> /* for atoi */
#include <stdio.h> /* for printf */

#include "stdlib.h" /* for atoi */
#include "stdio.h" /* for printf */

#include <string.h>

#define Vowel [aeiouy]

main () {
    yylex ();
    printf (#i, atoi(yytext) + 1);
}
```

A *context-free grammar* is a 4-tuple \((V, \Sigma, R, S)\), where we have:

- \(V\), a set of *variables* (or *non-terminals*)
- \(\Sigma\), a set of *terminals* such that \(V \cap \Sigma = \emptyset\)
- \(R\), a set of *rules*, where the LHS is a variable in \(V\) and the RHS is a string of variables in \(V\) and terminals in \(\Sigma\)
- \(S \in V\), the *start variable*

CFGs are stronger than regular expressions, and able to express recursively-defined constructs.

Example: we cannot write a regular expression for any number of matched parentheses:

```
( ), (()), ((())), ... 
```

Using a CFG:

```
E \rightarrow ( E ) | \epsilon 
```

**Automatic parser generators** use CFGs as input and generate parsers using the machinery of a deterministic pushdown automaton.

```
joo5.y
```

```
bison
```

```
y.tab.c
```

```
gcc
```

```
parser
```

```
AST
```

By limiting the kind of CFG allowed, we get efficient parsers.

Simple CFG example:

```
A \rightarrow a B 
A \rightarrow \epsilon 
B \rightarrow b B 
B \rightarrow c 
```

Alternatively:

```
A \rightarrow a B | \epsilon 
A \rightarrow \epsilon 
B \rightarrow b B | c 
```

In both cases we specify \(S = A\). Can you write this grammar as a regular expression?

We can perform a *rightmost derivation* by repeatedly replacing variables with their RHS until only terminals remain:

```
A 
 a B 
 a b B 
 a b b B 
 a b b c 
```
There are several different grammar formalisms. First, consider BNF (Backus-Naur Form):

\[
\text{stmt ::= stmt_expr ; } | \text{ while_stmt } | \text{ block } | \text{ if_stmt} \\
\text{while_stmt ::= WHILE (" expr ") stmt} \\
\text{block ::= "{ stmt_list "}} \\
\text{if_stmt ::= IF (" expr ") stmt | IF (" expr ") stmt ELSE stmt}
\]

We have four options for stmt_list:

1. \( \text{stmt_list ::= stmt_list stmt } \mid \epsilon \) \\
   \( \rightarrow 0 \) or more, left-recursive
2. \( \text{stmt_list ::= stmt stmt_list } \mid \epsilon \) \\
   \( \rightarrow 0 \) or more, right-recursive
3. \( \text{stmt_list ::= stmt_list stmt } \mid \text{stmt} \) \\
   \( \rightarrow 1 \) or more, left-recursive
4. \( \text{stmt_list ::= stmt stmt_list } \mid \text{stmt} \) \\
   \( \rightarrow 1 \) or more, right-recursive

Second, consider EBNF (Extended BNF):

\[
\begin{array}{|c|c|c|}
\hline
\text{BNF} & \text{derivations} & \text{EBNF} \\
\hline
A \rightarrow A \mid b & b A a & A \rightarrow \{ a \} \\
(\text{left-recursive}) & A a a & b a a \\
\hline
A \rightarrow a A \mid b & b a A & A \rightarrow \{ a \} b \\
(\text{right-recursive}) & a a A & a a b \\
\hline
\end{array}
\]

where ‘{’ and ‘}’ are like Kleene *'s in regular expressions. Using EBNF repetition, our four choices for stmt_list become:

1. \( \text{stmt_list ::= \{} \text{stmt} \} \)
2. \( \text{stmt_list ::= \{} \text{stmt} \} \)
3. \( \text{stmt_list ::= \{} \text{stmt} \} \) stmt
4. \( \text{stmt_list ::= stmt \{} \text{stmt} \} \)

EBNF also has an optional-construct. For example:

\[
\text{stmt_list ::= stmt stmt_list } \mid \text{stmt}
\]

could be written as:

\[
\text{stmt_list ::= stmt [ stmt_list ]}
\]

And similarly:

\[
\text{if_stmt ::= IF (" expr ") stmt | IF (" expr ") stmt ELSE stmt}
\]

could be written as:

\[
\text{if_stmt ::= IF (" expr ") stmt [ ELSE stmt ]}
\]

where ’[’ and ’]’ are like ‘?’ in regular expressions.
A grammar is ambiguous if a sentence has different parse trees:

\[ S 
\rightarrow S \; ; 
\]
\[ S 
\rightarrow id \; := \; E 
\]
\[ S 
\rightarrow print \; ( \; L \; ) 
\]

(a) \[ S := id := E \]

(b) \[ S := id := E + E \]

(c) \[ S := print \; ( \; L \; ) \]

A grammar is ambiguous if a sentence has different parse trees:

\[ id := id \; + \; id \; + \; id \]

The above is harmless, but consider:

\[ id := id \; - \; id \; - \; id \]

\[ id := id \; + \; id \; * \; id \]

Clearly, we need to consider associativity and precedence when designing grammars.
There are fundamentally two kinds of parser:

1) Top-down, *predictive* or *recursive descent* parsers. Used in all languages designed by Wirth, e.g. Pascal, Modula, and Oberon.

```
program
  const_decls type_decls proc_decls main
  const_decl const_decl ... local_decls body
  stmt stmt
```

One can (easily) write a predictive parser by hand, or generate one from an LL($k$) grammar:

- *Left-to-right parse*;
- *Leftmost-derivation*; and
- *$k$ symbol lookahead.*

Algorithm: look at beginning of input (up to $k$ characters) and unambiguously expand leftmost non-terminal.

---

The *shift-reduce* bottom-up parsing technique.

1) Extend the grammar with an end-of-file $\$$, introduce fresh start symbol $S'$:

```
S' \rightarrow S$
S \rightarrow S ; S    E \rightarrow id    L \rightarrow E
S \rightarrow id := E    E \rightarrow num    L \rightarrow L , E
S \rightarrow print ( L )    E \rightarrow E + E
                                     E \rightarrow ( S , E )
```

2) Choose between the following actions:

- **shift:**
  - move first input token to top of stack

- **reduce:**
  - replace $\alpha$ on top of stack by $X$
  - for some rule $X \rightarrow \alpha$

- **accept:**
  - when $S'$ is on the stack
Use a DFA to choose the action; the stack only contains DFA states now.

Start with the initial state (s1) on the stack.

Lookup (stack top, next input symbol):

- shift(n): skip next input symbol and push state n
- reduce(k): rule k is $X \rightarrow \alpha$; pop |\alpha| times; lookup (stack top, X) in table
- goto(n): push state n
- accept: report success
- error: report failure

```
0  S' \rightarrow S$
1  S \rightarrow S : S
2  S \rightarrow id := E
3  S \rightarrow print ( L )
4  E \rightarrow id
5  E \rightarrow num
6  E \rightarrow E + E
7  E \rightarrow ( S , E )
8  L \rightarrow E
9  L \rightarrow L , E
```
LR(1) is an algorithm that attempts to construct a parsing table:
- Left-to-right parse;
- Rightmost-derivation; and
- 1 symbol lookahead.

If no conflicts (shift/reduce, reduce/reduce) arise, then we are happy; otherwise, fix grammar.

An LR(1) item \( (A \rightarrow \alpha \cdot \beta \gamma, x) \) consists of:
1. A grammar production, \( A \rightarrow \alpha \beta \gamma \)
2. The RHS position, represented by ‘.’
3. A lookahead symbol, \( x \)

An LR(1) state is a set of LR(1) items.

The sequence \( \alpha \) is on top of the stack, and the head of the input is derivable from \( \beta \gamma x \). There are two cases for \( \beta \), terminal or non-terminal.

We first compute a set of LR(1) states from our grammar, and then use them to build a parse table. There are four kinds of entry to make:
1. goto: when \( \beta \) is non-terminal
2. shift: when \( \beta \) is terminal
3. reduce: when \( \beta \) is empty (the next state is the number of the production used)
4. accept: when we have \( A \rightarrow B \cdot $ \)

Follow construction on the tiny grammar:

Constructing the LR(1) NFA:
- start with state \( S \rightarrow \cdot E \$ \)
- state \( A \rightarrow \alpha \cdot B \beta \) has:
  - \( \epsilon \)-successor \( B \rightarrow \cdot \gamma \) if:
    * exists rule \( B \rightarrow \gamma \), and
    * \( x \in \text{lookahead}(\beta) \)
  - \( B \)-successor \( A \rightarrow \alpha B \cdot \beta \)
- state \( A \rightarrow \alpha \cdot x \beta \) has:
  - \( x \)-successor \( A \rightarrow \alpha x \cdot \beta \)

Constructing the LR(1) DFA:
Standard power-set construction, “inlining” \( \epsilon \)-transitions.
Conflicts

\[
\begin{align*}
A & \rightarrow B & x & \text{shift/reduce conflict} \\
A & \rightarrow C & x & \\
A & \rightarrow B & y & \text{shift/reduce conflict} \\
A & \rightarrow C & y & \\
A & \rightarrow B & x & \text{reduce/reduce conflict} \\
A & \rightarrow C & x & \\
\end{align*}
\]

\[
\begin{align*}
A & \rightarrow B & x & \rightarrow s_i \\
A & \rightarrow C & x & \rightarrow s_j \\
\end{align*}
\]

\text{shift/shift conflict?} \\
\Rightarrow \text{by construction of the DFA} \\
\text{we have } s_i = s_j

\text{LR(1) tables may become very large.}
\text{Parser generators use LALR(1), which merges states that are identical except for lookaheads.}

\text{bison (yacc)} is a parser generator:
- it inputs a grammar;
- it computes an LALR(1) parser table;
- it reports conflicts;
- it resolves conflicts using defaults (!); and
- it creates a C program.

Nobody writes (simple) parsers by hand anymore.

The grammar:

\[
\begin{align*}
1 & E \rightarrow \text{id} \\
4 & E \rightarrow E / E \\
7 & E \rightarrow ( E ) \\
2 & E \rightarrow \text{num} \\
5 & E \rightarrow E + E \\
3 & E \rightarrow E * E \\
6 & E \rightarrow E - E \\
\end{align*}
\]

is expressed in bison as:

\[
\%
/* C declarations */ \\
%
/* Bison declarations; tokens come from lexer (scanner) */ \\
%
%token tIDENTIFIER tINTCONST
%
%start exp
%
/* Grammar rules after the first % */ \\
%
exp : tIDENTIFIER \\
| tINTCONST \\
| exp '*' exp \\
| exp '/' exp \\
| exp '+' exp \\
| exp '-' exp \\
| '(' exp ')' \\
| \\
%
/* User C code after the second % */ \\
%
Input this code into exp.y to follow the example.
The grammar is ambiguous:

```
$ bison --verbose exp.y # --verbose produces exp.output
exp.y contains 16 shift/reduce conflicts.
```

```
$ cat exp.output
State 11 contains 4 shift/reduce conflicts.
State 12 contains 4 shift/reduce conflicts.
State 13 contains 4 shift/reduce conflicts.
State 14 contains 4 shift/reduce conflicts.
[...]

state 11
```

```
exp -> exp . '*' exp (rule 3)
exp -> exp '*' exp . (rule 3) <-- problem is here
exp -> exp . '/' exp (rule 4)
exp -> exp . '+' exp (rule 5)
exp -> exp . '-' exp (rule 6)
'/*' shift, and go to state 6
'//' shift, and go to state 7
'/*' shift, and go to state 8
'//' shift, and go to state 9
'/*' [reduce using rule 3 (exp)]
'//' [reduce using rule 3 (exp)]
'/*' [reduce using rule 3 (exp)]
'//' [reduce using rule 3 (exp)]
$default reduce using rule 3 (exp)
```

Rewrite the grammar to force reductions:

```
E -> E + T   T -> T * F   F -> id
E -> E - T   T -> T / F   F -> num
E -> T
```

```
%token tIDENTIFIER tINTCONST
%start exp

%%
exp : exp '+' term
   | exp '-' term
   | term

   term : term '*' factor
        | term '/' factor
        | factor

   factor : tIDENTIFIER
           | tINTCONST
           | '(' exp ')' ;

%%
```

```
which resolve shift/reduce conflicts:

Conflict in state 11 between rule 5 and token '+'
   resolved as reduce. <-- Reduce exp + exp . +
Conflict in state 11 between rule 5 and token '-'
   resolved as reduce. <-- Reduce exp + exp . -
Conflict in state 11 between rule 5 and token '//'
   resolved as reduce. <-- Reduce exp + exp . //
Conflict in state 11 between rule 5 and token '/*'
   resolved as shift.  <-- Shift exp + exp . */
Conflict in state 11 between rule 5 and token '/'
   resolved as shift.  <-- Shift exp + exp . /
```

Note that this is not the same state 11 as before.

Or use precedence directives:

```
%token tIDENTIFIER tINTCONST
%start exp

%left '+' '-'  /* left-associative, lower precedence */
%left '*' '/'  /* left-associative, higher precedence */

%%
exp : tIDENTIFIER
    | tINTCONST
    | exp '+' exp
    | exp '/' exp
    | exp '*' exp
    | exp '-' exp
    | '(' exp ')' ;

%%
```

The precedence directives are:

- `%left` (left-associative)
- `%right` (right-associative)
- `%nonassoc` (non-associative)

When constructing a parse table, an action is chosen based on the precedence of the last symbol on the right-hand side of the rule.

Precedences are ordered from lowest to highest on a linewise basis.

If precedences are equal, then:

- `%left` favors reducing
- `%right` favors shifting
- `%nonassoc` yields an error

This usually ends up working.
state 0
  tIDENTIFIER shift, and go to state 1
  tINTCONST shift, and go to state 2
  '(' shift, and go to state 3
  exp go to state 4

state 1
  exp -> tIDENTIFIER . (rule 1)
  $default reduce using rule 1 (exp)

state 2
  exp -> tINTCONST . (rule 2)
  $default reduce using rule 2 (exp)

state 14
  exp -> exp . '*' exp (rule 3)
  exp -> exp . '/' exp (rule 4)
  exp -> exp '/' exp . (rule 4)
  exp -> exp . '+' exp (rule 5)
  exp -> exp . '-' exp (rule 6)
  $default reduce using rule 4 (exp)

state 15
  $ go to state 16

state 16
  $default accept

$ cat exp.y
%
#include <stdio.h> /* for printf */
extern char *yytext; /* string from scanner */
void yyyerror() {
  printf("syntax error before %s\n", yytext);
}
%

union {
  int intconst;
  char *stringconst;
}

%token <intconst> tINTCONST
token <stringconst> tIDENTIFIER

%start exp
%left '+' '-'
%left '*' '/'

%%
[ 	
]+ /* ignore */;
"*" return '*';
"/" return '/';
"*" return '*';
"-" return '-';
"-" return '-';
"(" return '('.
")" return ')';
0|[(1-9)[0-9]+] { yylval.intconst = atoi (yytext);
  return tINTCONST;
}
[a-zA-Z_][a-zA-Z0-9_]* { yylval.stringconst =
  (char *) malloc (strlen (yytext) + 1);
  sprintf (yylval.stringconst, "%s", yytext);
  return tIDENTIFIER;
}
/* ignore */
%%

$ cat exp.l
%
#include "y.tab.h" /* for exp.y types */
#include <string.h> /* for strlen */
#include <stdlib.h> /* for malloc and atoi */
%
%%
[ 	
]+ /* ignore */;
"*" return '*';
"/" return '/';
"*" return '*';
"-" return '-';
"-" return '-';
"(" return '('.
")" return ')';
0|[(1-9)[0-9]+] { yylval.intconst = atoi (yytext);
  return tINTCONST;
}
[a-zA-Z_][a-zA-Z0-9_]* { yylval.stringconst =
  (char *) malloc (strlen (yytext) + 1);
  sprintf (yylval.stringconst, "%s", yytext);
  return tIDENTIFIER;
}
/* ignore */
%%

$ cat main.c
void yyparse();

int main (void)
{
  yyparse ();
}

Using flex/bison to create a parser is simple:
$ flex exp.l
$ bison --yacc --defines exp.y # note compatability options
$ gcc lex.yy.c y.tab.c y.tab.h main.c -o exp -lfl

When input a*(b-17) + 5/c:
$ echo "a*(b-17) + 5/c" | ./exp

our exp parser outputs the correct order of operations:
load a
load b
push 17
minus
push 5
load c
div
plus

You should confirm this for yourself!
If the input contains syntax errors, then the
bison-generated parser calls yyerror and stops.

We may ask it to recover from the error:

```c
exp : tIDENTIFIER { printf("load %s\n", $1); } 
  | '(': exp ')'
  | error { yyerror(); } 
; 
```

and on input `a@(b-17) ++ 5/c` get the output:

```
load a
syntax error before ( 
syntax error before ( 
syntax error before ( 
syntax error before b 
push 17 
minus 
syntax error before ) 
syntax error before ) 
syntax error before + 
plus 
push 5 
load c 
div 
plus
```

Error recovery hardly ever works.

---

SableCC (by Etienne Gagnon, McGill alumnus) is
a compiler compiler: it takes a grammatical
description of the source language as input, and
generates a lexer (scanner) and parser for it.

```
joos.sablecc
```

```
SableCC
```

```
foo.joos
```

```
joos/* java
```

```
javac
```

```
scanner parser
```

```
CST/AST
```

The SableCC 2 grammar for our Tiny language:

```
Package tiny;

Helpers
  tab = 9;
  cr = 13;
  lf = 10;
  digit = ['0'..'9'];
  lowercase = ['a'..'z'];
  uppercase = ['A'..'Z'];
  letter = lowercase | uppercase;
  idletter = letter | '_'
  idchar = letter | '_' | digit;

Tokens
  eol = cr | lf | cr lf;
  blank = ' ' | cr; 
  start = '+*';
  slash = '/';
  plus = '+';
  minus = '-';
  l_par = '(';
  r_par = ')';
  number = '0' [digit='0'] digit*;
  id = idletter idchar*;

Ignored Tokens
  blank, eol;

Productions
  exp = {plus} exp plus factor |
  {minus} exp minus factor |
  {factor} factor;
  factor = {mult} factor star term |
  {divd} factor slash term |
  {term} term;
  term = {paren} l_par exp r_par |
  {id} id |
  {number} number;
```

Version 2 produces parse trees, a.k.a. concrete
syntax trees (CSTs).
The SableCC 3 grammar for our Tiny language:

Productions

\[
\begin{align*}
\text{cst}\_\text{exp} (\rightarrow \text{exp}) &= \\
\quad (\text{cst}\_\text{plus}) \quad &\text{cst}\_\text{exp} \text{ plus factor} \\
\quad (\rightarrow \text{New} \text{exp.plus(cst}\_\text{exp}.\text{exp},\text{factor}\_\text{exp}.\text{exp})) \mid \\
\quad (\text{cst}\_\text{minus}) \quad &\text{cst}\_\text{exp} \text{ minus factor} \\
\quad (\rightarrow \text{New} \text{exp.minus(cst}\_\text{exp}.\text{exp},\text{factor}\_\text{exp}.\text{exp})) \mid \\
\quad (\text{factor}) \quad &\text{factor} (\rightarrow \text{factor}\_\text{exp})
\end{align*}
\]

\[
\begin{align*}
\text{factor} (\rightarrow \text{exp}) &= \\
\quad (\text{cst}\_\text{mult}) \quad &\text{factor} \text{ star term} \\
\quad (\rightarrow \text{New} \text{exp.mult(factor}\_\text{exp}.\text{exp},\text{term}\_\text{exp}.\text{exp})) \mid \\
\quad (\text{cst}\_\text{divd}) \quad &\text{factor} \text{ slash term} \\
\quad (\rightarrow \text{New} \text{exp.divd(factor}\_\text{exp}.\text{exp},\text{term}\_\text{exp}.\text{exp})) \mid \\
\quad (\text{term}) \quad &\text{term} (\rightarrow \text{term}\_\text{exp})
\end{align*}
\]

\[
\begin{align*}
\text{term} (\rightarrow \text{exp}) &= \\
\quad (\text{paren}) \quad &l\_\text{par} \text{ cst}\_\text{exp} r\_\text{par} (\rightarrow \text{cst}\_\text{exp}) \mid \\
\quad (\text{cst}\_\text{id}) \quad &\text{id} (\rightarrow \text{New} \text{id(id)}) \mid \\
\quad (\text{cst}\_\text{number}) \quad &\text{number} (\rightarrow \text{New} \text{number(number)})
\end{align*}
\]

Abstract Syntax Tree

\[
\begin{align*}
\text{exp} &= \\
\quad (\text{plus}) \quad &l\_\text{exp} [r]\_\text{exp} \mid \\
\quad (\text{minus}) \quad &l\_\text{exp} [r]\_\text{exp} \mid \\
\quad (\text{mult}) \quad &l\_\text{exp} [r]\_\text{exp} \mid \\
\quad (\text{divd}) \quad &l\_\text{exp} [r]\_\text{exp} \mid \\
\quad (\text{id}) \quad &\text{id} \mid \\
\quad (\text{number}) &\text{number};
\end{align*}
\]

Version 3 generates abstract syntax trees (ASTs).